Operating Systems Design

Synchronization

Instructor: Michela Becchi

Recall

- Processes/threads can execute concurrently
  - May be interrupted at any time
- Process/thread scheduling is non-deterministic
- Processes can compete for resources or cooperate
  - Processes/threads may be unaware of each other

- Concurrent access to shared data may result in data inconsistency
- Data consistency requires mechanisms to ensure the orderly execution of processes/threads

Producer-Consumer problem

Producer thread

```
while (true) {
   while (counter == BUFFER_SIZE)
      ; // do nothing
   /* produce an item and put
      it in the buffer */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer thread

```
while (true) {
   while (counter == 0)
      ; // do nothing
   /* extract an item from the
      buffer */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
}
```

Where is the problem?

Producer-Consumer problem (cont'd)

Producer thread

```
while (true) {
   while (counter == BUFFER_SIZE)
      ; // do nothing
   /* produce an item and put
      it in the buffer */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer thread

```
while (true) {
   while (counter == 0)
      ; // do nothing
   /* extract an item from the
      buffer */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
}
```

Race condition

Race condition

- `counter++` can be implemented as
```
register1 = counter
register1 = register1+1
counter = register1
```
- `counter --` can be implemented as
```
register2 = counter
register2 = register2-1
counter = register2
```
- Consider the following interleaving (assume initially `counter=5`)

Producer	Consumer	Values
`register1=counter`		
`register1=register1+1`		
	`register2=counter`	
	`register2=register2-1`	
`counter=register1`		
	`counter=register2`	

Race condition

- `counter++` can be implemented as
```
register1 = counter
register1 = register1+1
counter = register1
```
- `counter --` can be implemented as
```
register2 = counter
register2 = register2-1
counter = register2
```
- Consider the following interleaving (assume initially `counter=5`)

Producer	Consumer	Values
`register1=counter`		`register1=5`
`register1=register1+1`		`register1=6`
	`register2=counter`	`register2=5`
	`register2=register2-1`	`register2=4`
`counter=register1`		`counter=6`
	`counter=register2`	`counter=4`

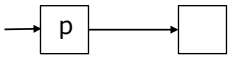
Race condition (cont'd)

- When does it occur?
  - Multiple processes/threads read and write shared data concurrently
    - Concurrency, shared data, interference
  - The final result depends on the particular order of execution
- Why?
  - Operating system scheduling is non-deterministic
  - Many possible process/thread execution interleavings
- For correct operation:
  - Execution result must be independent of interleaving

More examples

- Concurrent access to shared linked list

```
a) n->next=p->next          1) m->next=p->next
b) p->next=n                2) p->next=m
```



- What would be the outcome of each code in isolation?
- What is the outcome of interleaving `ab12`
- What is the outcome of interleaving `a12b`

- Concurrent access to shared device (e.g., printer)

Critical section problem

- Given *n* threads $\{T_0, T_1, \ldots T_{n-1}\}$
- Each thread has critical section segment of code which
  - modifies common variable, updates shared data structures, writes shared files, etc.
  - contains race condition
  - cannot be executed by multiple threads concurrently
- Critical section problem
  - design protocol allowing the threads to cooperate

Critical section solution

- Mutual exclusion
  - No two processes/threads can be executing in their critical section at the same time
- Progress
  - If no process/thread is executing in its critical section and there are some processes/threads ready to enter their critical section, the selection of the process/thread that will enter the critical section next cannot be postponed indefinitely
- Bounded waiting
  - A bound must exist on the number of times that other processes/ threads are allowed to enter their critical sections after a process/thread has made a request to enter its critical section and before that request is granted

Critical section handling in OS

- Preemptive kernels
  - allow preemption of processes when running in kernel mode
    - may be more responsive
    - may be more suitable to real-time programming
- Nonpreemptive kernels
  - Do not allow preemption of processes running in kernel mode
    - a kernel-mode process will run until it exits kernel mode, blocks or voluntarily yield control of the CPU
    - free from race conditions on kernel data structures

Mutual exclusion – hardware support

- Uniprocessor systems
  - Interrupt disabling while shared variables are being modified

```
while(true){
        /* disable interrupts */
        /* critical section */
        /* enable interrupts */
        /* remainder section */
}
```

- What about multiprocessor systems?

Mutual exclusion – hardware support

- Implement locking through hardware instructions that execute more actions (e.g., read & write) atomically
  - `compare_and_swap()`
  - `test_and_set()`
  - `exchange()`
- General idea of locks

```
while(true){
        /* acquire lock */
        /* critical section */
        /* release lock */
        /* remainder section */
}
```


Mutual exclusion – hardware support

```
int compare_and_swap(int *value, int expected, int new_value) {
   int old_value = *value;
   if (old_value == expected)
      *value = new_value;
   return old_value;
}
```

Executed atomically

```
while(true) {
   while (compare_and_swap(&lock, 0, 1) == 1)
      ; // do nothing
   /* critical section */
   lock = 0; // release lock
   /* remainder section */
}
```

Spinlock => busy waiting

Hardware support – pros & cons

- Advantages
  - Simple and easy to verify
  - Applicable to any number of processes (on single processor or multiple processors) sharing main memory
  - Multiple critical sections
- Disadvantages
  - Busy waiting
  - Starvation
    - Some process might wait indefinitely
  - Deadlock
    - What happens if process in critical section is interrupted?

Mutual exclusion with bounded waiting

```
while (true) {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = compare_and_swap(&lock, false, true);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;              BOUNDED WAITING – wait of at most n-1 turns
   while ((j != i) && !waiting[j]) // find the first process that is
      j = (j + 1) % n;             // waiting
   if (j == i)                    // the lock is released only if there
      lock = false;               // is no process waiting
   else
      waiting[j] = false;         // => process j won't need to acquire
                                  //    the lock
   /* remainder section */
}
```

Mutual exclusion – software support

Semaphore

- Integer variable
- Three atomic operations
  - initialization to value ≥ 0
  - `semWait` (`P, wait, sem_wait, acquire`)
    - decrements semaphore
    - If value becomes < 0, calling process is blocked (it goes on a wait)
  - `semSignal` (`Q, signal, sem_post, release`)
    - increments semaphore
    - if value ≤ 0 and there are waiting processes, wake up one of them
- Invariant
  - If ≤ 0, the value is equal to the number of waiting processes

Semaphores implementation w/ spinlock

```
semWait(S) {
   while (S <= 0)
      ; /* busy wait */
   S--;
}
```

```
semSignal(S) {
    S++;
}
```

Semaphores implementation w/ queue

- Semaphore = integer variable + queue

```
semWait(S) {
   S--;
   if (S < 0){
      /* place this process in queue
         and block it */
    }
}
```

```
semSignal(S) {
    S++;
    if (S <= 0){
       /* remove a process from the queue
          and place it in the ready list */
    }
}
```


Semaphores (cont'd)

- Counting semaphores
  - any integer value
- Binary semaphores
  - Can only take values 0 or 1
    - `semWait` blocks current process if value=0, decrements if value=1
    - `semSignal` unblocks process only if value=0
  - Easy to implement
  - Similar to mutex lock
    - Mutex lock requires the process that locks the mutex to also unlock it

Semaphores for ordering

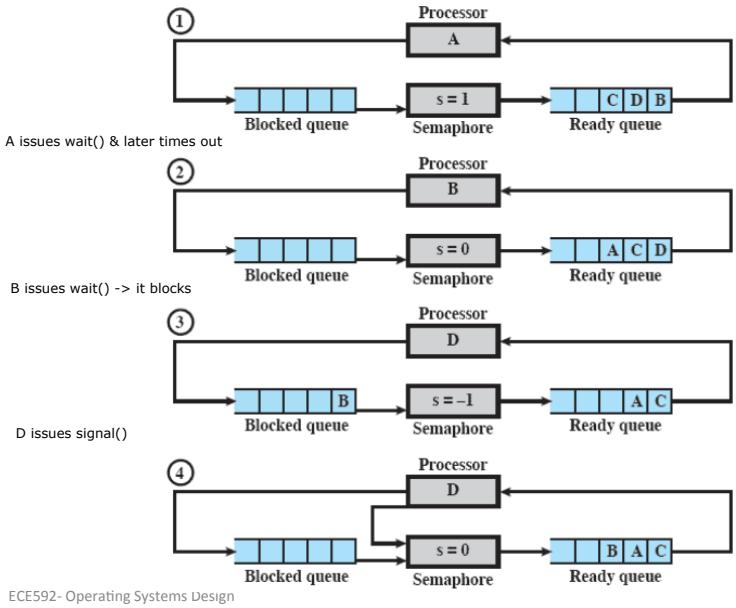
```
sem_t s;

/* child */
void *child (void *arg){
   /* some code … */
   sem_signal(&s); // Pthreads semSignal
   return NULL;
}

/* parent */
int main(…){
   sem_init(&s, …); //semaphore initialization
   pthread_t c;
   pthread_create(c, NULL, child, NULL);
   sem_wait(&s); //Pthreads semWait
   /* some code based on data produced by child … */
   return 0;
}
```

- What is the initialization value of the semaphore?
  - what if parent issues `sem_wait` before/after child issues `sem_signal`?

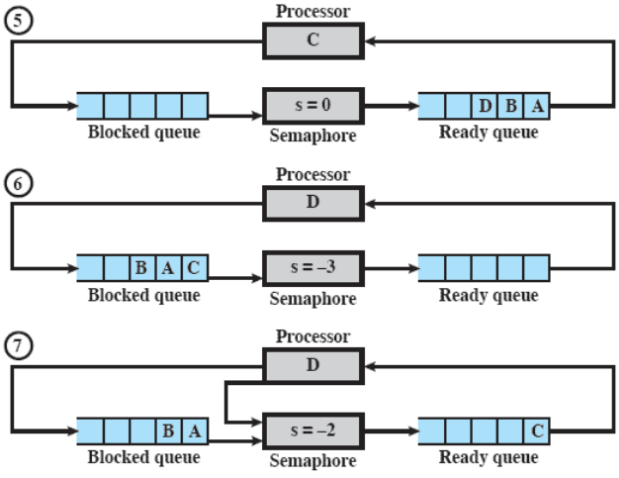
Semaphore implementation

- Must guarantee that no two processes can execute `semSignal` and `semWait` on the same semaphore at the same time
- This is a critical section problem
  - Single-processor: inhibit interrupts during `semSignal` and `semWait`
  - Multiprocessor: use spinlocks or hardware primitives (e.g., `compare_and_swap`)
- Implementation using a queue to hold processes waiting on a semaphore
  - Strong semaphore: queue handled in FIFO fashion
  - Weak semaphore: unspecified order of removal from the queue

Strong semaphore example



Strong semaphore example (cont'd)



Semaphores in XINU

- Definition:
  - `include/semaphore.h`
- Implementation: in `system`
  - `semcreate.c`
  - `semdelete.c`
  - `semcount.c`
  - `semreset.c`
  - `wait.c`
  - `signal.c`

Software support – pros & cons

- Advantages
  - Simple and easy to verify?
  - Applicable to any number of processes sharing main memory?
  - Multiple critical sections?
- Disadvantages
  - Busy waiting?
  - Starvation?
  - Deadlock?

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
  - Let *s* and *Q* be two semaphores initialized to 1

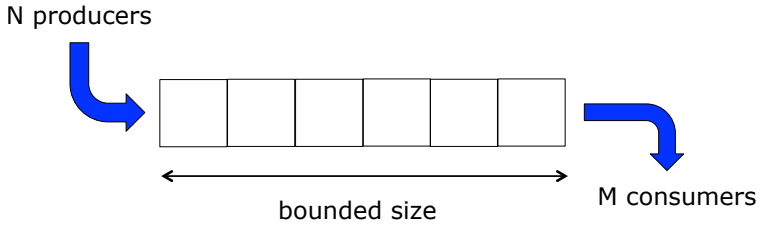
P_0	P_1
`semWait(S);`	`semWait(Q);`
`semWait(Q);`	`semWait(S);`
`...`	`...`
`semSignal(S);`	`semSignal(Q);`
`semSignal(Q);`	`semSignal(S);`

- Starvation – A process may never be removed from the semaphore queue in which it is suspended

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-buffer problem



- producers cannot write to full buffer
- consumers cannot read from empty buffer

Bounded-buffer problem w/ semaphores

```
Item buffer[n];
int in=0, out=0;
Semaphore mutex=1; // controls access to the buffer
Semaphore empty=n; // number of free buffer entries
Semaphore full=0;  // number of occupied buffer entries
```

```
void insert(Item item){
   semWait(empty);
   semWait(mutex);

   /* add item to the buffer */
   buffer[in]=item;
   in = (in+1) % N;

   semSignal(mutex);
   semSignal(full);
}
```

PRODUCER

```
Item remove(){
      ???
      ???

   /* remove item from buffer */

      ???
   return item;
}
```

CONSUMER

What if we have only 1 producer and 1 consumer?

Bounded-buffer problem w/ semaphores (cont'd)

What if we swap `mutex` and `full/empty` operations?

```
void insert(Item item){
   semWait(mutex);
   semWait(empty);

   /* add item to the buffer */
   buffer[in]=item;
   in = (in+1) % N;

   semSignal(full);
   semSignal(mutex);
}
```

PRODUCER

```
Ttem remove(){
   semWait(mutex);
   semWait(full);

   /* remove item from buffer */
   item = buffer[out];
   out = (out+1) % N;

   semSignal(empty);
   semSignal(mutex);
   return item;
}
```

CONSUMER

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set
  - Writers – can both read and write
- Where is the problem?
  - Can multiple readers access the shared data at the same time?
  - What about writers?
- Several variations:
  - First reader-writers problem – readers have priority
  - Second reader-writer problem – writers have priority

First Readers-Writers Problem

```
int readers=0; //number of active readers
Semaphore rw_mutex=1;
Semaphore mutex=1;
```

```
do {
   semWait(rw_mutex);
   ...
   /* writing is performed */
   ...
   semSignal(rw_mutex);
} while (true);
```

WRITERS

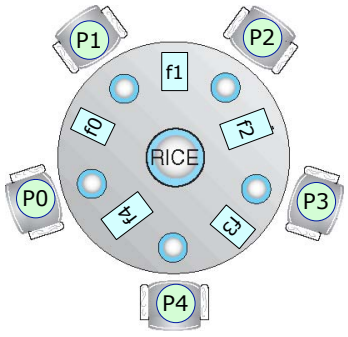
```
do {
   semWait(mutex);
   readers++;
   if (readers == 1)
      semWait(rw_mutex);
   semSignal(mutex);
   ...
   /* reading is performed */
   ...
   semWait(mutex);
   readers--;
   if (readers == 0)
      semSignal(rw_mutex);
   semSignal(mutex);
} while (true);
```

READERS

- Busy waiting?
- Starvation?
- Deadlock?

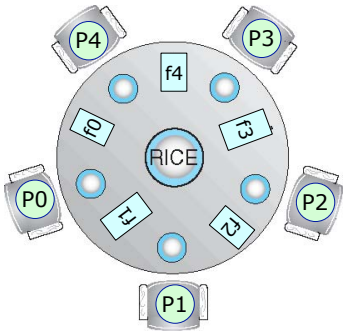
Dining-Philosophers Problem (Dijkstra)

- 5 philosophers spend their lives alternating thinking and eating
- In order to eat, they need to pick up 2 forks (one at a time)
  - Need both to eat, then release both when done
- Solution with
  - Concurrency
  - No starvation
  - No deadlock



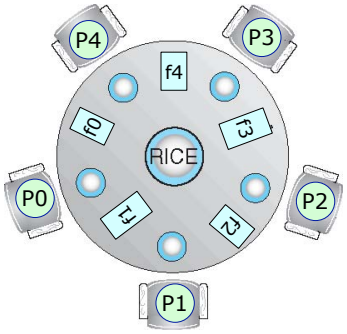
Dining-Philosophers Problem –cont'd

```
while (true){
   think();
   getforks();
   eat();
   putforks();

}

int left(int p){return p;}
int right(int p) {return (p+1)%5;}
```



Dining-Philosophers Problem – cont'd

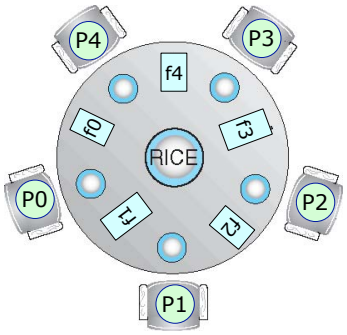
```
Semaphore forks[5];

void getforks(){
   semWait(forks[left(p)]);
   semWait(forks[right(p)]);
}

void putforks(){
   semSignal(forks[left(p)]);
   semSignal(forks[right(p)]);
}
```



- Problems?

Dining-Philosophers Problem – cont'd

```
Semaphore forks[5];

void getforks(){
   if(p!=4){
      semWait(forks[left(p)]);
      semWait(forks[right(p)]);
   }else{
      semWait(forks[right(p)]);
      semWait(forks[left(p)]);
   }
}
```



Condition Variables

- Problem: a thread may want to wait for a condition to hold before proceeding
- Condition variable
  - Explicit queue that a thread can put itself on while waiting for a condition to become true
  - two operations:
    - `wait()` – puts a thread to sleep while waiting for a condition to become true
    - `signal()` – wakes up other threads waiting on a condition

Pthreads primitives (mutexes & cond. var.)

- Mutexes
  - `int pthread_mutex_lock(pthread_mutex_t *mutex)`
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex)`
- Condition variables
  - `int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`
    - Assumes mutex `m` is locked when `pthread_cond_wait` is invoked
    - Releases lock when putting thread to sleep
    - Re-acquires lock before returning to caller
  - `int pthread_cond_signal(pthread_cond_t *c)`
    - unblocks at least one thread waiting on condition `c`
    - no effect if there are no threads blocked on condition `c`
  - `int pthread_cond_broadcast(pthread_cond_t *c)`
    - unblocks all the threads waiting on condition `c`
    - no effect if there are no threads blocked on condition `c`

Example - parent waiting for child

```
volatile int done = 0;

/* child */
void *child (void *arg){
   /* some code … */
   done = 1;
   return NULL;
}

/* parent */
int main(…){
   pthread_t c;
   pthread_create(c, NULL, child, NULL);
   while(done==0);
   /* some code based on data produced by child … */
   return 0;
}
```

Problems?

Example - parent waiting for child (cont'd)

```
volatile int done = 0;
pthread_mutex_t m = …; //mutex initialization
pthread_cond_t c = …;  //cond. var. initialization
```

```
/* parent */
int main(…){
   pthread_t c;
   pthread_create(c, NULL, child, NULL);
   pthread_mutex_lock(&m);
   while (done == 0)
      pthread_cond_wait(&c, &m);
   pthread_mutex_unlock(&m);
   /* some code based on data produced by
   child … */
   return 0;
}
```

```
/* child */
void *child (void *arg){
   /* some code … */
   pthread_mutex_lock(&m);
   done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_lock(&m);
   return NULL;
}
```


Example - parent waiting for child (cont'd)

What if we do not use the `done` variable?

```
pthread_mutex_t m = …; //mutex initialization
pthread_cond_t c = …;  //cond. var. initialization
```

```
/* parent */
int main(…){
   pthread_t c;
   pthread_create(c, NULL, child, NULL);
   pthread_mutex_lock(&m);
   pthread_cond_wait(&c, &m);
   pthread_mutex_unlock(&m);
   /* some code based on data produced by
   child … */
   return 0;
}
```

```
/* child */
void *child (void *arg){
   /* some code … */
   pthread_mutex_lock(&m);
   pthread_cond_signal(&c);
   pthread_mutex_lock(&m);
   return NULL;
}
```

Example - parent waiting for child (cont'd)

What if we do not hold a lock in order to `signal`/`wait`?

```
volatile int done = 0;
pthread_cond_t c = …;  //cond. var. initialization
```

```
/* parent */
int main(…){
   pthread_t c;
   pthread_create(c, NULL, child, NULL);
   if (done == 0)
        pthread_cond_wait(&c, &m);
   /* some code based on data produced by
   child … */
   return 0;
}
```

```
/* child */
void *child (void *arg){
   /* some code … */
   done = 1;
   pthread_cond_signal(&c);
   return NULL;
}
```


Checking conditions: `while` or `if`?

- `while` is always correct; `if` might be depending on the semantics of signaling
- Spurious wakeups – with some packages two or more threads can be waken up by a single signal
  - `while` allows rechecking the condition

Bounded-buffer problem w/ Cond. Var.

```
Item buffer[N];
int in=0, out=0;
int count = 0;
pthread_mutex_t m;
pthread_cond_t cond;
```

What if 1 producer & 1 consumer?
What if multiple producers/consumers (e.g. 1 producer & 2 consumers)?

```
void insert(Item item){
   pthread_mutex_lock(&m);
   if (count==N)
     pthread_cond_wait(&cond,&m);

   /* add item to the buffer */
   buffer[in]=item;
   in = (in+1) % N;
   count++;

   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&m);
}
```

PRODUCER

```
Item remove(){
   pthread_mutex_lock(&m);
   if (count==0)
     pthread_cond_wait(&cond,&m);

   /* remove item from buffer */
   item = buffer[out];
   out = (out+1) % N;
   count --;

   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&m);
   return item;
}
```

CONSUMER

Bounded-buffer problem w/ Cond. Var.

```
Item buffer[N];
int in=0, out=0;
int count = 0;
pthread_mutex_t m;
pthread_cond_t cond;
```

MESA semantics: signaling wakes up a thread, but does not guarantee that the thread will run immediately (and that the state won't change before it will run)

```
void insert(Item item){
   pthread_mutex_lock(&
   if (count==N)
     pthread_cond_wait(

   /* add item to the bu
   buffer[in]=item;
   in = (in+1) % N;
   count++;

   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&m);
}
```

PRODUCER

```
   item = buffer[out];
   out = (out+1) % N;
   count --;

   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&m);
   return item;
}
```

CONSUMER

Bounded-buffer problem w/ Cond. Var.

```
Item buffer[N];
int in=0, out=0;
int count = 0;
pthread_mutex_t m;
pthread_cond_t cond;
```

what if the buffer has only 1 element?

what if the buffer has only N elements?

```
void insert(Item item){
   pthread_mutex_lock(&m);
   while (count==N)
     pthread_cond_wait(&cond,&m);

   /* add item to the buffer */
   buffer[in]=item;
   in = (in+1) % N;
   count++;

   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&m);
}
```

PRODUCER

```
It
   while (count==0)
     pthread_cond_wait(&cond,&m);

   /* remove item from buffer */
   item = buffer[out];
   out = (out+1) % N;
   count --;

   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&m);
   return item;
}
```

CONSUMER

Bounded-buffer problem w/ Cond. Var.

```
Item buffer[N];
int in=0, out=0;
int count = 0;
pthread_mutex_t m;
pthread_cond_t empty, full;
```

```
void insert(Item item){
   pthread_mutex_lock(&m);
   while(count==N)
     pthread_cond_wait(&empty,&m);

   /* add item to the buffer */
   buffer[in]=item;
   in = (in+1) % N;
   count++;

   pthread_cond_signal(&full);
   pthread_mutex_unlock(&m);
}
```

PRODUCER

```
Item remove(){
   pthread_mutex_lock(&m);
   while(count==0)
     pthread_cond_wait(&full,&m);

   /* remove item from buffer */
   item = buffer[out];
   out = (out+1) % N;
   count --;

   pthread_cond_signal(&empty);
   pthread_mutex_unlock(&m);
   return item;
}
```

CONSUMER

Covering conditions

```
int bytesLeft = MAX_HEAP_SIZE;
pthread_cond_t c;
pthread_mutex_t m;
```

```
void *allocate(int size) {
  pthread_mutex_lock(&m);
  while (bytesLeft < size)
    pthread_cond_wait(&c, &m);
  void *ptr = ...; // get mem from heap
  bytesLeft -= size;
  pthread_mutex_unlock(&m);
  return ptr;
}
```

```
void free(void *ptr, int size) {
  pthread_mutex_lock(&m);
  bytesLeft += size;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

`bytesLeft = 0`

T_1: `allocate(100)`

T_2: `allocate(10)`

T_3: `free(50)`

problem?