Process Synchronization

8. Process synchronization and solutions
23. Lock (aka mutex)
28. Producer & Consumer Problem
33. Semaphore

ECE469

Jan 21

Yiying Zhang

Review of Lec3

- What is a process?
- Benefits of process isolation?
- Process state transition diagram
- What is PCB? What are stored in it?

- Why concurrent processes?
- Process creation and termination
  - fork()
  - exec()

Lec1: Policies vs. Mechanisms

When fork() creates a child process

- Execution possibilities?
  - Parent and child execute concurrently
  - Parent waits till child finishes
    - Concurrency?

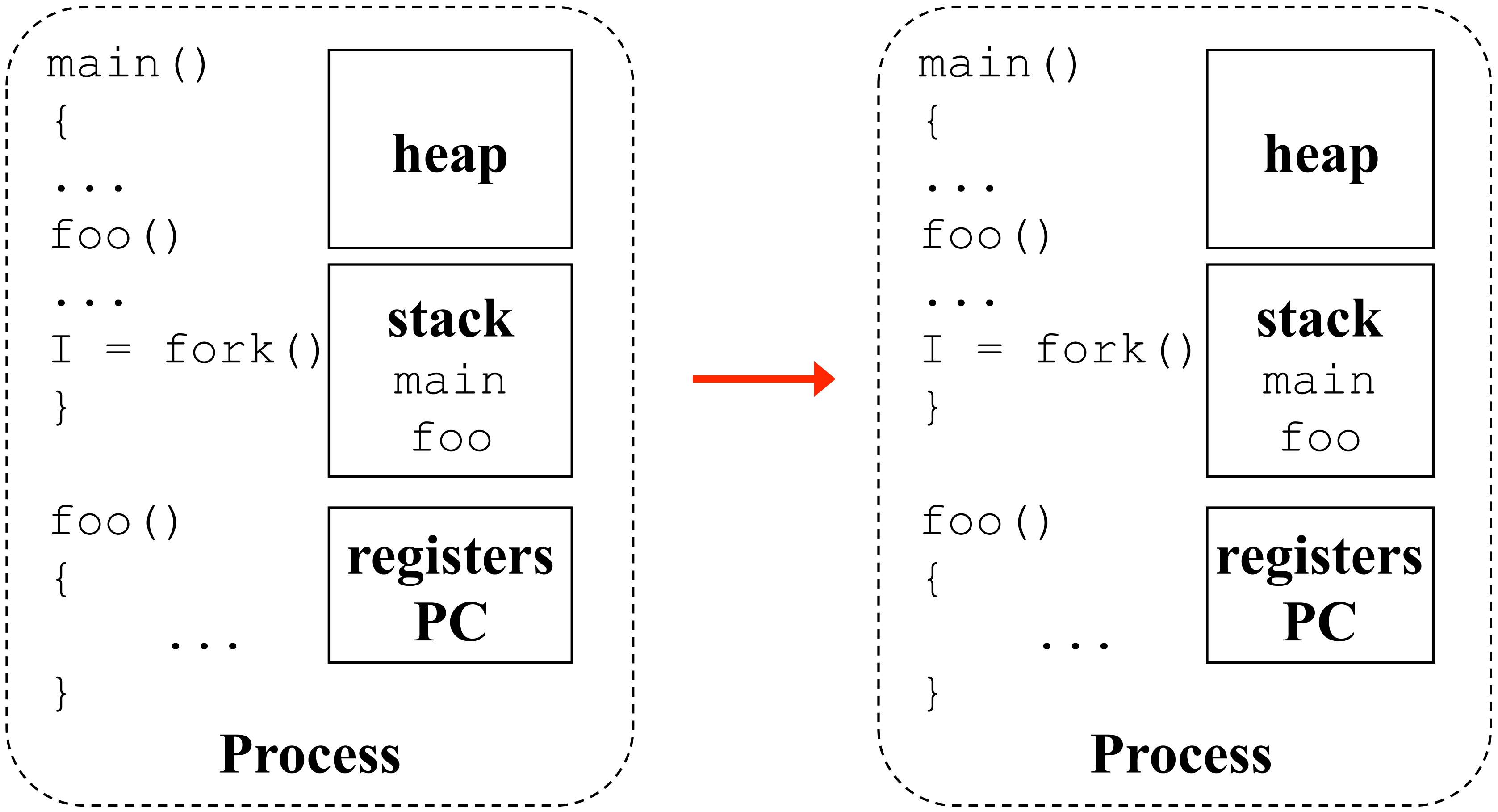
- Address space possibilities?
  - Child duplicates that of parent
  - Child has a program loaded into it

[lec3] Process Creation –- UNIX examples

- *fork()* system call creates a duplicate of the original process
  - Should have been called "clone()"
  - Allows the two to communicate (one time)
  - How to disambiguate who is who?

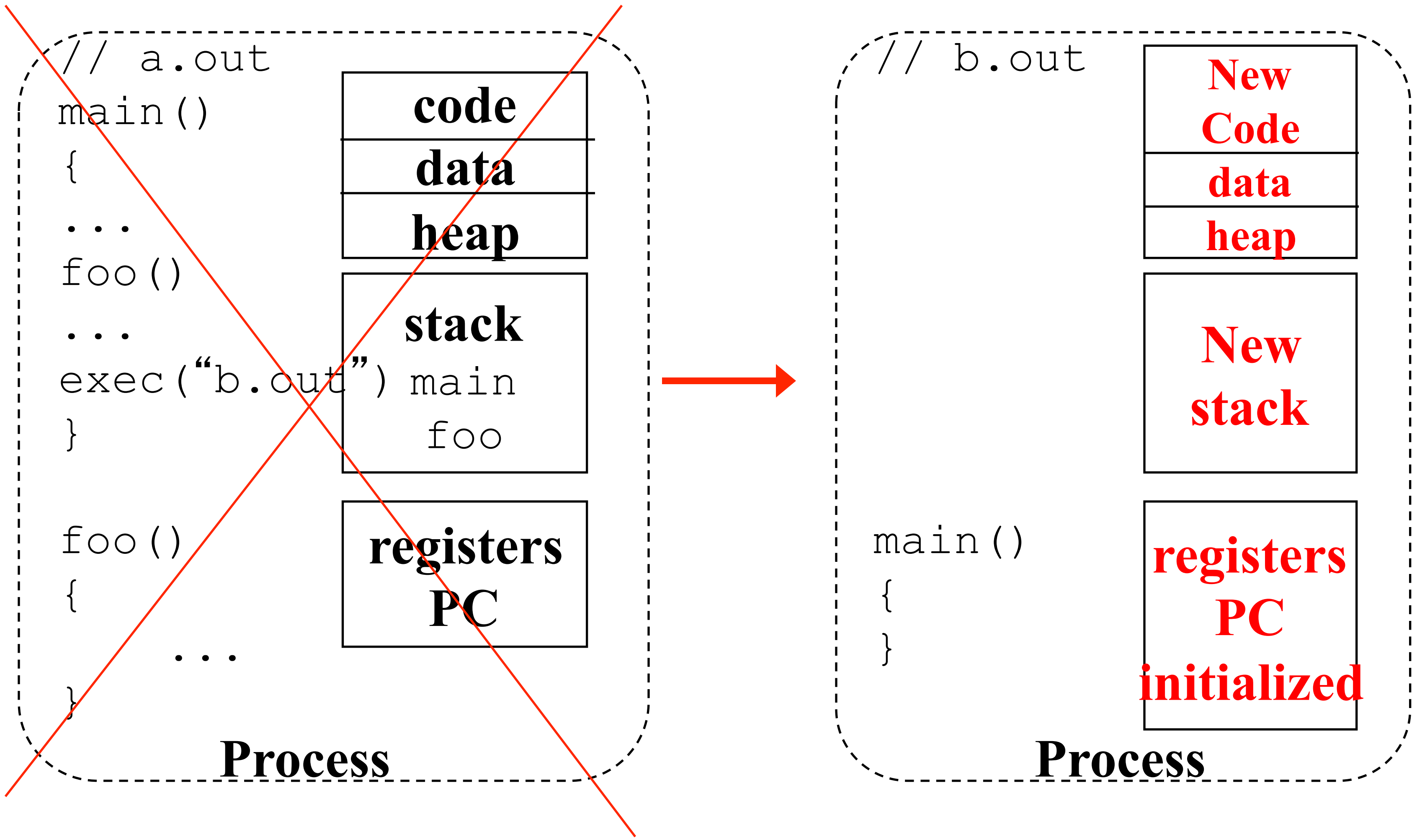
- *exec()* system call used after a *fork* to replace the process' code/address space with a new program

fork()



main()
{
...
foo()
...
I = fork()
}
foo()
{
...
}
heap
stack
main
foo
registers
PC
Process
main()
{
...
foo()
...
I = fork()
}
foo()
{
...
}
heap
stack
main
foo
registers
PC
Process

exec("b.out")



Afterwards, only one thing about the process was kept, which is?

Reading assignment

- Dinosaur Chapter 6 (process synchronization)
  - Read up materials covered
- Comet Chapters 26, 28, 29
  - Related to concurrency and synchronization

Process synchronization

- Cooperating processes may share data via
  - shared address space (code, data, heap) by using threads
  - shared memory objects (used in lab2)
  - Files
  - (Sending messages)
- What can happen if processes try to access shared data (address) concurrently?
  - Sharing bank account with sibling:

    At 3pm: If (balance > $10) withdraw $10
- How hard is the solution?

What is a Good Solution?

- Correctness: Only one process inside a critical section (mutual exclusion)
  - No assumption about CPU speeds
  - Works for multiprocessors
- Performance
  - Processes outside of critical section should not block other processes (progress)
  - No one waits forever (bounded waiting)
- Ease of programming
  - Symmetric code for all

How to achieve mutual exclusion?

- Need a *locking* mechanism

- Elements of locking
  - must lock before using
  - must unlock after done
  - must wait if locked

“Too Much Milk” Problem

Person A	Person B
Look in fridge: out of milk	
Leave for Meijer	
	Look in fridge: out of milk
Arrive at Meijer	
	Leave for Meijer
Buy milk	
	Arrive at Meijer
Arrive home	
	Buy milk
	Arrive home

- How to put in a locking mechanism?

A Possible Solution?

```
if ( noMilk ) {
   if (noNote) {
      leave note;
      buy milk;
      remove note;
   }
}
```

```
if ( noMilk ) {
   if (noNote) {
      leave note;
      buy milk;
      remove note;
   }
}
```

- Process can get context switched after checking milk and note, but before leaving note
- Why does it work for human?

Why does it work for people?

- Human can perform *test* (look for other person & milk) and *set* (leave note) at the same time.

Another Possible Solution?

process A

```
leave noteA
if (noNoteB) {
  if (noMilk) {
    buy milk
  }
}
remove noteA
```

process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

- process A switched out right after leaving a note

“too much milk” Yet Another Possible Solution?

process A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

- Safe to buy
- If the other buys, quit
- Things we dislike this solution?

Remarks

- The last solution works, but
  - life is too complicated
  - A’s code is different from B’s
  - busy waiting is a waste
- What we want is:

```
Acquire(lock);
if (noMilk)        }
  buy milk;        }  Critical Section
Release(lock);
```

Process synchronization needs help from OS and hardware!

Mutual exclusion & Critical Section

- *Critical section* – a section of code, or collection of operations, in which only one process shall be executing at a given time

- *Mutual exclusion* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

Example of Critical Section

- Concurrent accesses to shared variables, at least one of which is write

P0:	P1:	P2:
`Read note;`	`Read note;`	`write note;`
…	…	…

Desirable properties of MuEx

- *Fair*: if several processes are waiting, let each in eventually
- *Efficient*: don't use up substantial amounts of resources when waiting (e.g. no busy waiting)
- *Simple*: should be easy to use (e.g. just bracket the critical sections)

Desirable properties of processes using MuEx

- Always lock before manipulating shared data
- Always unlock after manipulating shared data

- Do not lock again if already locked
- Do not unlock if not locked by you

- Do not spend large amounts of time in critical section

Mutual Exclusion provided by OS (and language/compiler)

- Locks
  - Alone can solve simple problems
- More advanced
  - Semaphore
  - Lock and condition variable
    - Lock alone is not flexible enough
  - Monitor
- Each primitive itself is atomic!

Lock (aka mutex)

Init: lock = 1; // 0 means held; 1 means free

lock_acquire(lock)

```
{
    while (lock==0)
        ;
    lock--;
}
```

lock_release(lock)

```
{
    if (lock == 0)
        lock++;
}
```

- Each primitive is atomic
- In reality, lock is not implemented as above!
    - The waiting process is put to sleep

"Too much milk" problem with locks

```
Acquire(lock);
if (noMilk)
  buy milk;
Release(lock);
```

$\}$ Critical Section

- What is the problem with this solution?

Deep thinking

- How can we separate "checking" from "buying milk" and only lock "checking"?

```
Lock_flag = FALSE;

Acquire(lock);
if (no note && noMilk){
  leave note;
  local_flag = true; }
Release(lock);

If (local_flag) buy milk;
Acquire(lock)
If (local_flag){
   local_flag = FALSE;
   remove note;}
Release (lock);
```

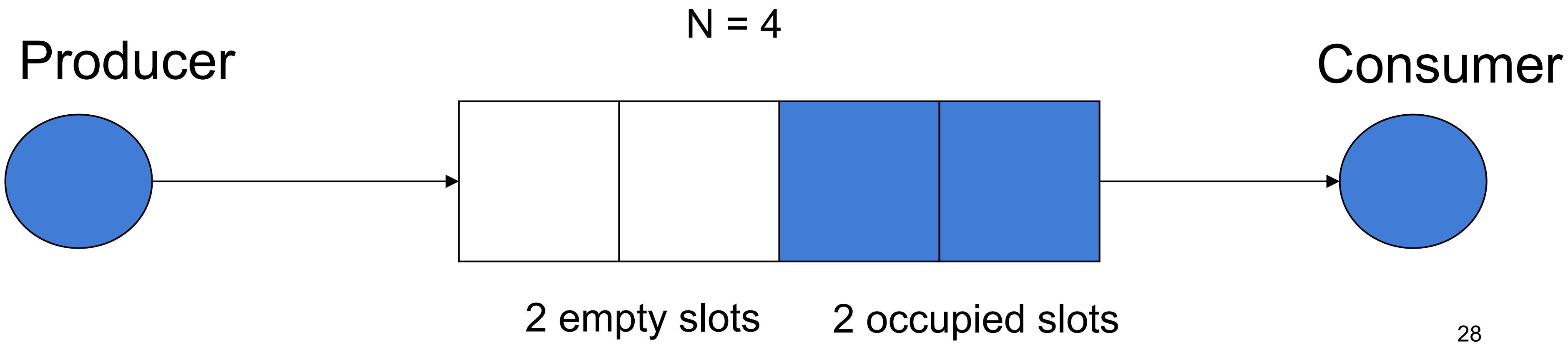
Break – One of the hardest math-24 puzzle

- How do you apply +, -, *, /, (,) on 3, 3, 8, 8 to get 24? Each number can only be used once. For example with 2, 2, 8, 8, we have (2 + 2) * 8 - 8 = 24. However, 3, 3, 8, 8 is considerably harder.

Often times, we have to wait for shared resources

Producer & Consumer Problem

- Producer: creates copies of a resource
- Consumer: uses up (destroys) copies of a resource.
- Buffers: fixed size, used to hold resource produced by producer before consumed by consumer



Producer & Consumer Problem

- Synchronization: keeping producer & consumer in sync

- Happens inside OS all the time (e.g., I/Os)
  - How about in real life?

Producer & Consumer

Producer

```
while (1) {

 produce an item;

 while (pool is full)
           ;

 insert(item to pool)

}
```

Consumer

```
While (1) {

while (pool is empty)
          ;

remove(item from pool);

consume the item;

}
```

Producer & Consumer -- Locks

Producer

```
while (1) {

 produce an item;

 while (pool is full);

  acq(lock);
 insert(item to pool)
  rel(lock);

}
```

Consumer

```
While (1) {

 while (pool is empty);

  acq(lock);
 remove(item from pool);
  rel(lock);

 consume the item;

}
```

Often times, we have to wait for shared resources

- Busy waiting is a bad idea
- Checking resources itself needs to be in critical section
- Busying waiting inside CS even worse!
  - No one else can check!

→ Need a more powerful sync. primitive!

→ Want the simplest primitive that can check & wait

Semaphore

- A synchronization variable that takes on non-negative integer values
  - Invented by Edsger Dijikstra in the mid 60's s

- Two primitve operations
  - wait(semaphore): an atomic operation that waits for semaphore to become greater than 0, then decrements it by 1
  - signal(semaphore): an atomic operation that increments semaphore by 1

Semaphore

```
wait(S) {
    while (S<=0)
        ;
    S--;
}
```

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

- Historically, wait() is known as P(), signal is known as V();
- In reality, P(S) is not implemented as above
- Semaphore aren’t provided by hardware (why not?) – we’ll discuss OS implementations next time

Semaphores Definition (Dijkstra)

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). *You cannot read the current value of the semaphore.*

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore. <= 0 block; 变成1， take the lock

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

- ACK: "The Little Book of Semaphores" by Allen B. Downey
  - Available online (free) at: http://www.greenteapress.com/semaphores/

Binary Semaphore

Init: S =1;

```
wait(S) {
    while (S==0)
        ;
    S--;
}
```

```
signal(S) {
    if (S == 0)
        S++;
}
```

- Binary semaphores: only take 0 or 1
- Sounds familiar?
  - S=0 → someone is holding the lock!

semaphores vs. locks: fundamental difference?

Semaphores

integer

```
wait(S) {
    while (S<=0);
    S--;
}
```

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

Binary Semaphore (lock) 0/1

```
wait(S) {
    while (S==0);
    S--;
}
```

```
signal(S) {
    if (S == 0) S++;
}
```

Q: why do we need to check S==0 for Binary Semaphore?

Semaphore

P(S)

```
wait(S) {
    while (S<=0);
    S--;
}
```

V(S)

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

What happens if initially S = 1

- P1: P(S), …, V(S)
- P2: P(S), …, V(S)
- P3: P(S), …, V(S)

Semaphore

P(S)

```
wait(S) {
    while (S<=0);
    S--;
}
```

V(S)

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

What happens if initially S = 1

- P1: P(S), …, V(S)
- P2: P(S) --------→, …, V(S)
- P3: P(S) ----------------------------→, …, V(S)

Semaphore

P(S)

```
wait(S) {
    while (S<=0);
    S--;
}
```

V(S)

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

What happens if initially S = 1

- P1: P(S), …, V(S)
- P2: V(S), …, P(S)
- P3: V(S), …, P(S)

Semaphore

P(S)

```
wait(S) {
    while (S<=0);
    S--;
}
```

V(S)

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

What happens if initially S = 2

- P1: P(S), …, V(S)
- P2: P(S), …, V(S)
- P3: P(S), …, V(S)

Semaphore

P(S)

```
wait(S) {
    while (S<=0);
    S--;
}
```

V(S)

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

What happens if initially S = 2

- P1: P(S), …, V(S)
- P2: P(S), …, V(S)
- P3: P(S)---------→, …, V(S)

semaphore has built-in counting!

- signal(S) simply increments S
  - "just produced an item"
  - S value = how many items have been produced

- wait(S) will return without waiting only if S > 0;
  - Wait(S) is saying "waited until there is at least one item, and just consumed an item"

Two usages of semaphores

- For mutual exclusion:
  - to ensure that only one process is accessing shared info at a time.
  - Semaphores or binary semaphores?
- For condition synchronization:
  - to permit processes to wait for certain things to happen
  - Semaphores or binary semaphores?

Producer & Consumer (cont)

- Define constraints (what is "correct")
  - Consumer must wait for producer to fill buffers (mutual excl. or condition sync?)
  - Producer must wait for consumer to empty buffers, if all buffer space is in use
    (mutual excl. or condition sync?)
- Use a separate semaphore for each constraint
  - Full = 0
  - Empty = N

Producer & Consumer – semaphore attempt, what’s wrong?

Producer

```
while (1) {

  produce an item;

  wait(EMPTY);

  insert(item to pool);

  signal(FULL)
}
```

Consumer

```
While (1) {

  wait(FULL);

   remove(item from pool);

  sginal(EMPTY);

  consume the item;
}
```

Init: FULL = 0; EMPTY = N;

Look Closely

```
int buffer[MAX];
int fill = 0;
int use = 0;

insert (int value) {
        buffer[fill] = value;
        fill = (fill + 1) % MAX
}
```

```
int get() {
        int tmp = buffer[use]
        use = (use + 1) % MAX
        return tmp;
}
```

Need to protect shared resource
(critical section) !

Producer & Consumer (cont)

- Define constraints (what is “correct”)
  - Consumer must wait for producer to fill buffers (mutual excl. or condition sync?)
  - Producer must wait for consumer to empty buffers, if all buffer space is in use
    (mutual excl. or condition sync?)
  - Only one process must manipulate buffer pool at once (mutual excl. or condition sync?)

- Use a separate semaphore for each constraint
  - Full = 0
  - Empty = N
  - Mutex = 1

Producer & Consumer – semaphore attempt 2, what's wrong?

Producer

```
while (1) {

  produce an item;

  acq(lock);
  wait(EMPTY);

  insert(item to pool);

  signal(FULL)
  rel(lock);
}
```

Consumer

```
While (1) {

  acq(lock);
  wait(FULL);

  remove(item from pool);
  sginal(EMPTY);
  rel(lock);

 consume the item;
}
```

Init: FULL = 0; EMPTY = N; Mutex = 1;

Deadlock!

Producer & Consumer – semaphore working

Producer

```
while (1) {

  produce an item;

  wait(EMPTY);

  acq(lock);
  insert(item to pool);
  rel(lock);

  signal(FULL)
}
```

Consumer

```
While (1) {

  wait(FULL);

  acq(lock);
  remove(item from pool);
  rel(lock)

  sginal(EMPTY);

  consume the item;
}
```

Init: FULL = 0; EMPTY = N; Mutex = 1;

Producer & Consumer – 2 pools

Producer

```
while (1) {

    get empty bucket  from buffer
     of empties

    produce data in bucket

    add full bucket to buffer of fulls

}
```

Consumer

```
While (1) {

    get full bucket from buffer of
     fulls

    consume data in buffer

    add empty bucket to buffer of
            empties

}
```

Producer & Consumer (2 pools)

Producer

```
while (1) {

    while (no empty buffer in Pool
     of Es)
          ;

    get empty buffer  from pool of
         empties

    produce data in buffer

    add full buffer to pool of fulls
}
```

Consumer

```
While (1) {

    while (no full buffer in pool of
     Fs)
          ;

    get full buffer  from pool of  fulls

    consume data in buffer

    add empty buffer to pool of
         empties
}
```

Producer & Consumer (2 pools) – solution using semaphores

Producer

```
while (1) {
  wait(EMPTY);
  acq(lock);
  get empty buffer  from pool of
       empties
  rel(lock);
 produce data in buffer
 acq(lock);
 add full buffer to pool of fulls
  rel(lock);
  signal(FULL);
 }
```

FULL=0; EMPTY=N; MUTEX = 1;

Consumer

```
  While (1) {
    wait(FULL);
    acq(lock);
    get full buffer  from pool of
         fulls
    rel(lock);
    consume data in buffer
    acq(lock);
    add empty buffer to pool of
        empties
    rel(lock);
    signal(EMPTY);
}
```

Producer & Consumer (2 pools) – wrong solution using semaphores

Producer

```
while (1) {
  acq(lock);
  wait(EMPTY);
  get empty buffer  from pool of
      empties
  rel(lock);
  produce data in buffer
  acq(lock);
  add full buffer to pool of fulls
  rel(lock);
  signal(FULL);
}
```

deadlock

Consumer

```
While (1) {
  wait(FULL);
  acq(lock);
  get full buffer  from pool of
      fulls
  rel(lock);
  consume data in buffer
  acq(lock);
  add empty buffer to pool of
      empties
  rel(lock);
  signal(EMPTY);
}
```

FULL=0; EMPTY=N; MUTEX = 1;

Deep thinking (take home)

- Why do we need to put mutex around accessing shared data

Text

- Why does producer wait(empties) but signal(fulls)?
  - Explain in terms of creating/destroying resources

- Is the order of wait()'s important?
- Is the order of signal()'s important?

- Could we have separate mutex semaphores for each pool?
- How would this be extended to > 1 consumers?

Producer & Consumer (2 pools) – lock solution? (take home)

Producer

```
while (1) {
 retry:
   acq(lock);
   if (no empty buffer in Pool of Es)
      {rel(lock); goto retry);}
   else
      get empty buffer  from pool of empties
   rel(lock);

   produce data in buffer

   acq(lock);
   add full buffer to pool of fulls
   rel(lock);
}
```

releasing lock, if using while loop,
then no one can get lock in consumer

Consumer

```
While (1) {
   acq(lock);
   while (no full buffer in pool of
      Fs);
   rel(lock);
   get full buffer  from pool of
      fulls

   consume data in buffer

   acq(lock);
   add empty buffer to pool of
      empties
   rel(lock);
}
```

keep checking without releasing