Electrical Engineering and Computer Science

EECS 358 - INTRODUCTION TO PARALLEL COMPUTING

Lecture 4

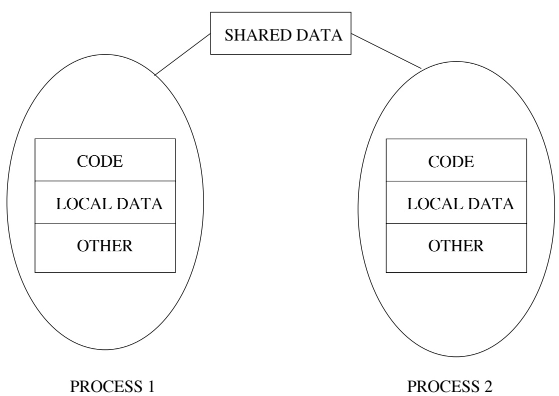
Shared Memory Programming I

Outline

- An overview of shared memory parallel programming
- Process model: creation and destruction
- Shared variables
- Synchronization: locks, barriers
- READING: Kumar, Chapter 7

Processes

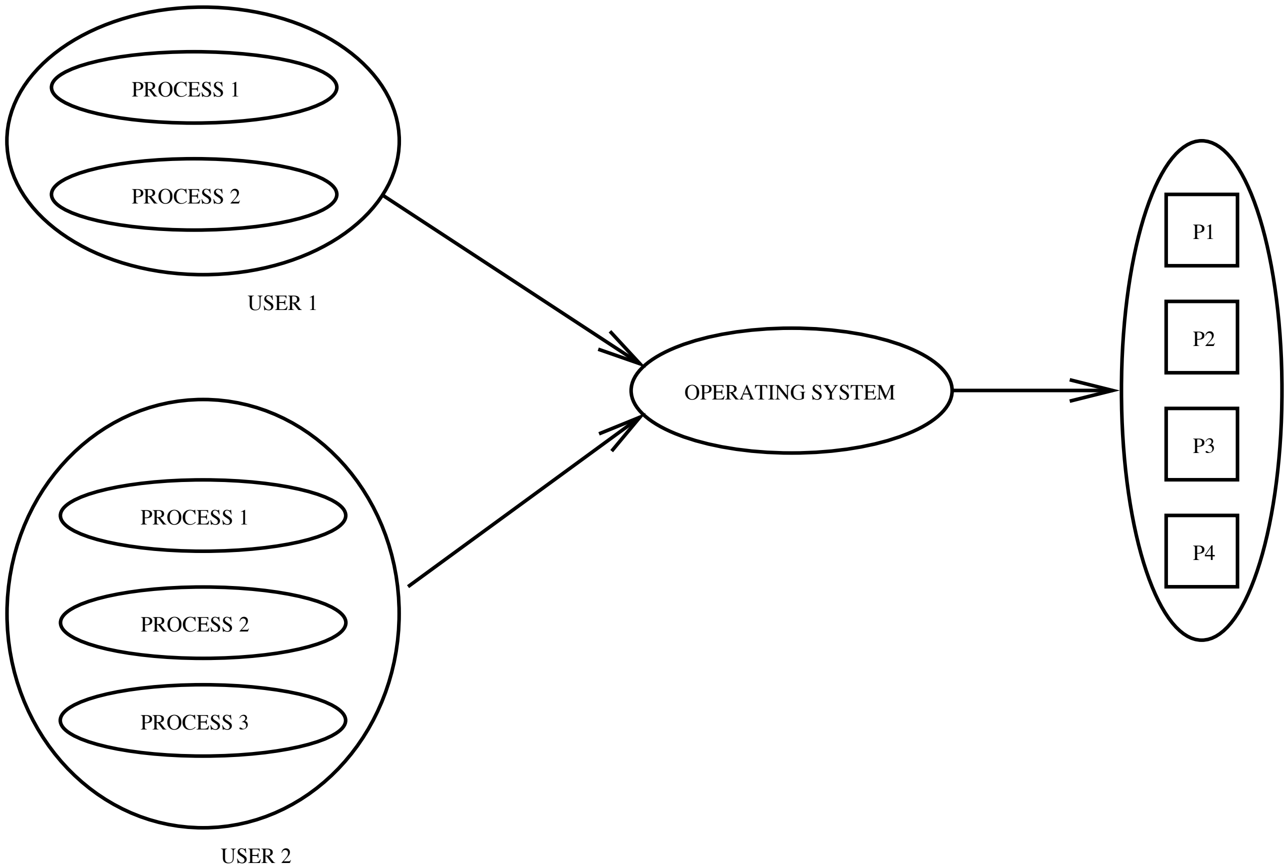
- Processes are central to shared memory programming
- A process is a program along with all of its enviroment (support structures)
- A set of processes can share variables among themselves and therefore communicate
- All processes have a unique ID associated with them



Processes

- Processes are created at the user's request by the operating system
- Processes are managed entirely by the operating system
- Processes are mapped to the processors in the system by the operating system
- Mapping of processes to processors is not under user control - it is assumed to be random for the purposes of parallel programming

Processes


PROCESS 1
PROCESS 2
USER 1
PROCESS 1
PROCESS 2
PROCESS 3
USER 2
OPERATING SYSTEM
P1
P2
P3
P4

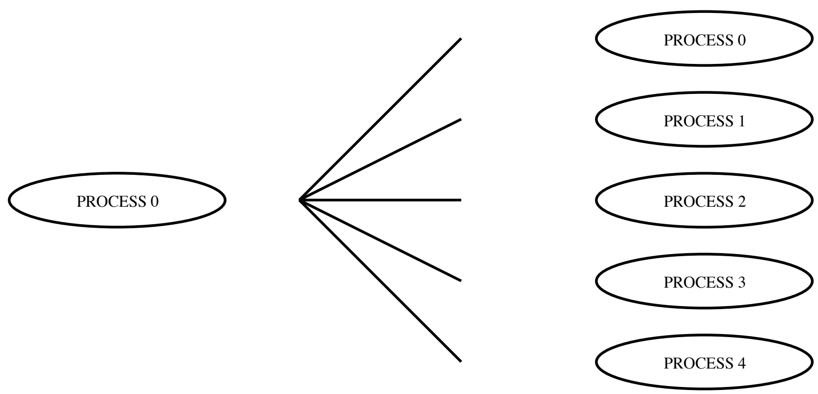
Processes

- Program execution is started using a single process (parent process)
- This process can create/destroy additional processes during program execution (child processes)
- Program computation is shared among parent and child processes:
  - Data parallel or function parallel
  - Coarse-grain or fine-grain
- Parallel execution achieved through mapping the processes to available physical processors in the system

Process Creation

- We will illustrate parallel programming calls using SGI IRIX operating system, other machines similar
- Use a process creation primitive:

```
C calling conventions
m_set_procs(nproc);
m_fork(func,[args]);
```



Process Creation

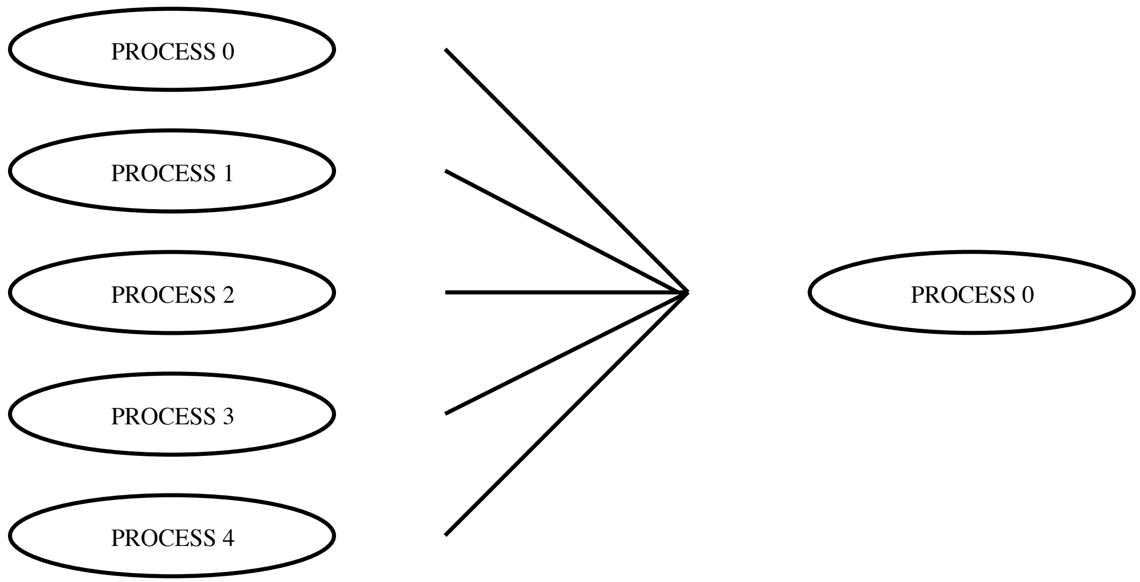
- Each child process is an exact copy of its parent
- All the variables associated with the parent are private for the parent and each child unless explicitly made shared
- The parent has ID=0, children share the other nproc-1 IDs

```
id = m_get_myid();
nprocs = m_get_procs();
```

Process Destruction

- Use a process destruction primitive

```
m_kill_procs();
```



Process Destruction

- All child processes are terminated; only the parent remains
- The parent waits for all child processes to terminate before returning

Example

```
#include <ulocks.h>
#include <task.h>

int i;

main() {
  m_set_procs(2);
  m_fork(func);
  m_kill_procs();
}

void func() {
  int id;

  id=m_get_myid();
  printf("%d - hello world\n",id);
}
```

OUTPUT:

```
1 - hello world
0 - hello world
```

Process Blocking and Unblocking

- Process creation and destruction can be very expensive. On the other hand, keeping processes idle is also expensive
- Alternative - put processes to sleep when not needed and revive them later:

```
main() {

m_fork(parallel_func);
m_kill_procs()
}
void parallel_func() {

(PARALLEL SECTION)
m_park_procs();
(SERIAL SECTION)
m_rele_procs()
(PARALLEL SECTION)
}
```

Shared Variables

- All global variables are shared between processes:

```
foo xxx,yyy,zzz;

main() {
   ...
}
void parallel_func() {
   ...
}
```

- Variables can also be shared using parameter passing

Example

```
float sum,sum0,sum1;

main() {

  m_set_procs(2);
  m_fork(parallel_func);
  m_kill_procs();

  sum=sum0+sum1;
  printf("total sum is %lf\n",sum);
}

void parallel_func() {

  int id;

  id=m_get_myid()
  if (id==0) sum0=1.0+2.0;
  else sum1=3.0+4.0;
}
```

Contention

- Random process scheduling creates problems with respect to modifying shared variables

- The final values of a shared variable can differ from run to run depending on the order in which it was modified

- Root of this contention problem is the simultaneous accessing of a shared variable

- Solution to the contention problem is to provide primitives for the restricted access of shared variables

Example

```
float total_sum;

main() {
  total_sum=0.0;
  m_set_procs(2);
  m_fork(parallel_func);
  m_kill_procs();
  printf("total sum is %f\n",total_sum);
}

void parallel_func() {
  int id;
  float partial_sum;

  id = m_get_myid();
  if (id==0) partial_sum=1.0+2.0;
  else partial_sum=3.0+4.0;

  total_sum=total_sum + partial_sum;
}
```

Example

OUTPUT:

```
total sum is 10.0
(ts = 0;
P0: ts = ts(0) + 3;
P1: ts = ts(3) + 7;)
```

```
total sum is 3.0
(ts = 0;
P1: ts = ts(0) + 7;
P0: ts = ts(0)+ 3;)
```

```
total sum is 7.0
(ts = 0;
P0: ts = ts(0)+ 3;
P1: ts = ts(0) + 7;)
```

- First output correct because of no contention
- Other outputs wrong because of contention - both processors read the same value of total_sum (0.0) and one or the other processor updates it last

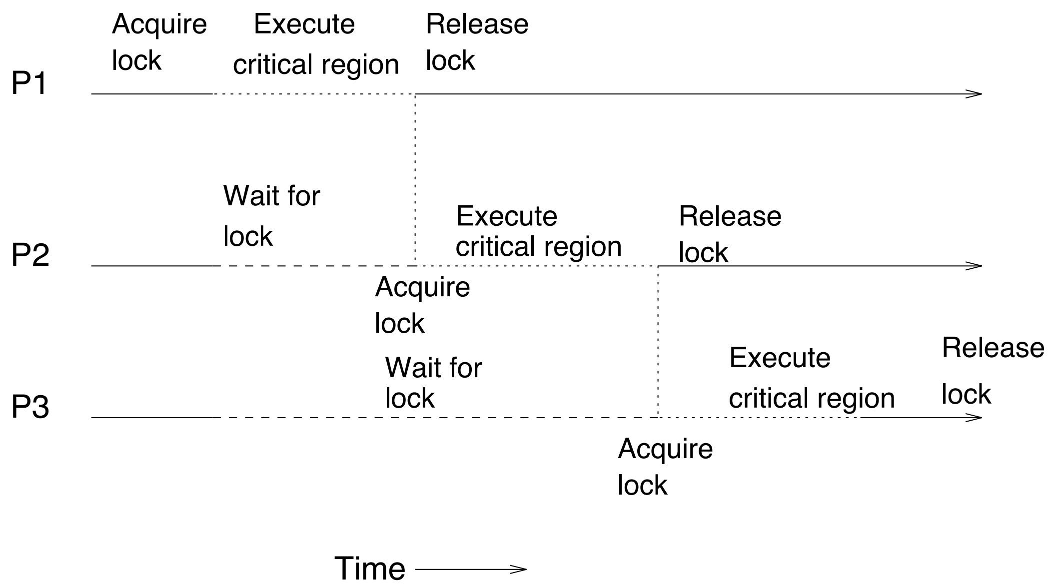
Variable Locks

- Locks are used to provide exclusive access for the modification of a variable
- A lock variable is to be created for every shared variable that needs exclusive access:
  - It must be shared among all processes that need to use it
  - It can either be in a "locked" or "unlocked" state
  - A locked state indicates ongoing exclusive modification of the variable the lock corresponds to
  - An unlocked state indicates no ongoing modification of the variable

Variable Locks

- The steps to be followed for exclusive access using locks are:
  - Acquire the lock for a variable - first, ensure the lock is in the unlocked state and then switch it to the locked state
  - Modify the variable - read its value and write a new value
  - Release the lock for the variable - switch the lock to the unlocked state so that others may have access
- The use of locks sequentializes execution of program sections and hurts performance

Illustration of Locks



Example

```
float total_sum;

void parallel_func() {
  int id;
  float partial_sum;

  if (id==0) partial_sum=1.0+2.0;
  else partial_sum=3.0+4.0;

  m_lock();
    total_sum+=partial_sum;
  m_unlock();
}
```

Race Conditions

```
float total_sum;

void parallel_func() {
  int id;
  float partial_sum, average;

  if (id==0) partial_sum=1.0+2.0;
  else partial_sum=3.0+4.0;

  m_lock();
    total_sum+=partial_sum;
  m_unlock();

  average=total_sum/4.0;
  printf("%d average is %f\n",id,average);
}
```

OUTPUT:

```
0 average is 2.5
1 average is 2.5
```

```
1 average is 1.75
0 average is 2.5
```

```
0 average is 0.75
1 average is 2.5
```

Barriers

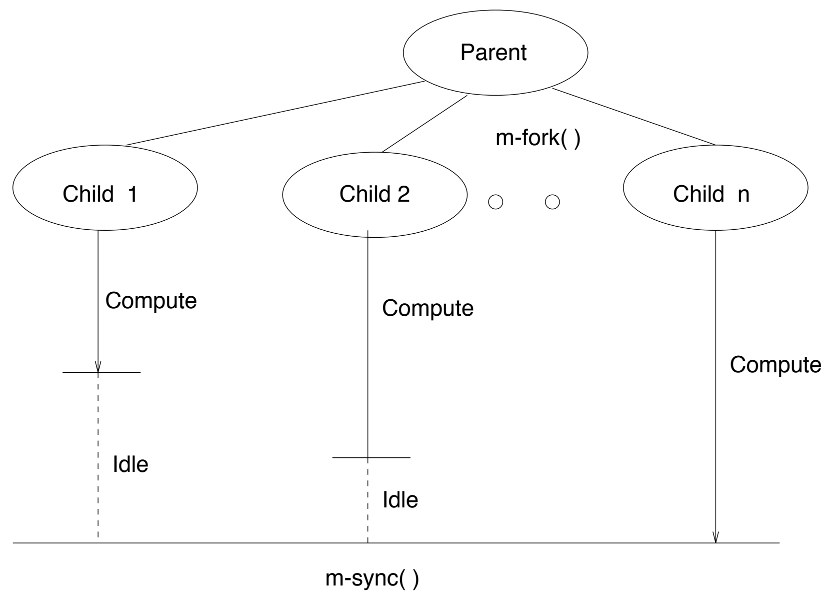
- A race condition exists if the results of a parallel program depend on the relative execution speed of processes

- Barriers enable processes to synchronize with each other and help avoid race conditions

- When a process enters a barrier, it waits for all other processes involved to reach the barrier before continuing

- Barriers cause performance degradation and therefore must be used judiciously

Illustration of Barriers



Barriers

```
float total_sum;

void parallel_func() {
  int id;
  float partial_sum, average;

  if (id==0) partial_sum=1.0+2.0;
  else partial_sum=3.0+4.0;

  m_lock();
    total_sum+=partial_sum;
  m_unlock();

  m_barrier();

  average=total_sum/4.0;
  printf("%d average is %f\n",id,average);
}
```

Process Summary

- Process creation/destruction
- Shared variables for inter-process communication
- Locks for exclusive access to shared variables
- Barriers to avoid race conditions

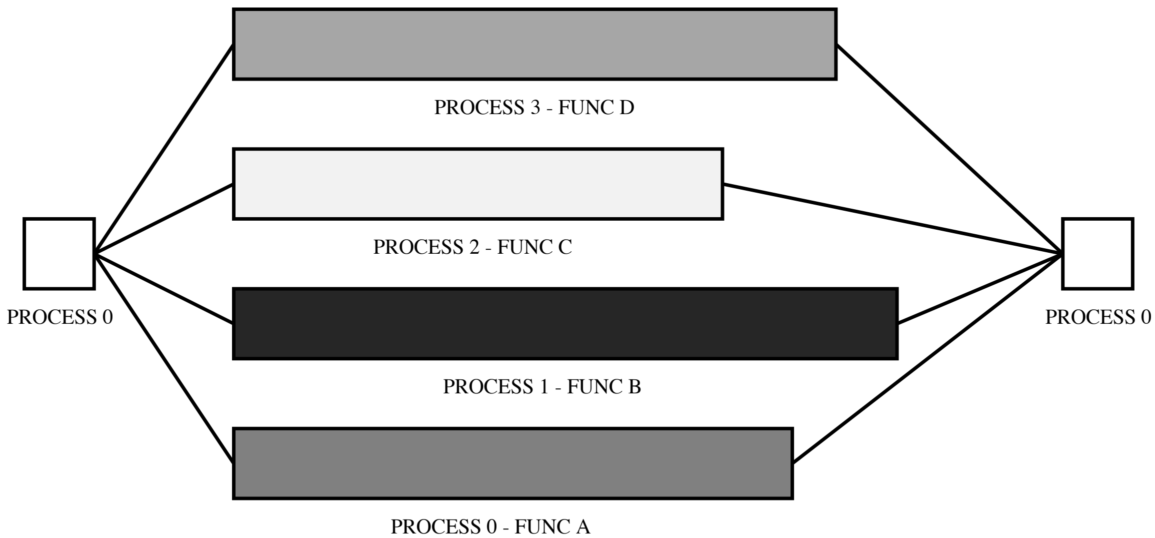
Process Based Parallel Program Model

- Processes executing in a function parallel fashion

```
main() {
  m_set_procs(4);
  m_fork(func);
  m_kill_procs();
}

void func() {
  switch (id) {
    case 0 : funcA();
     break;
    case 1 : funcB();
     break;
    case 2 : funcC();
     break;
    case 3 : funcD();
     break;
  }
}
```

Process Based Parallel Program Model

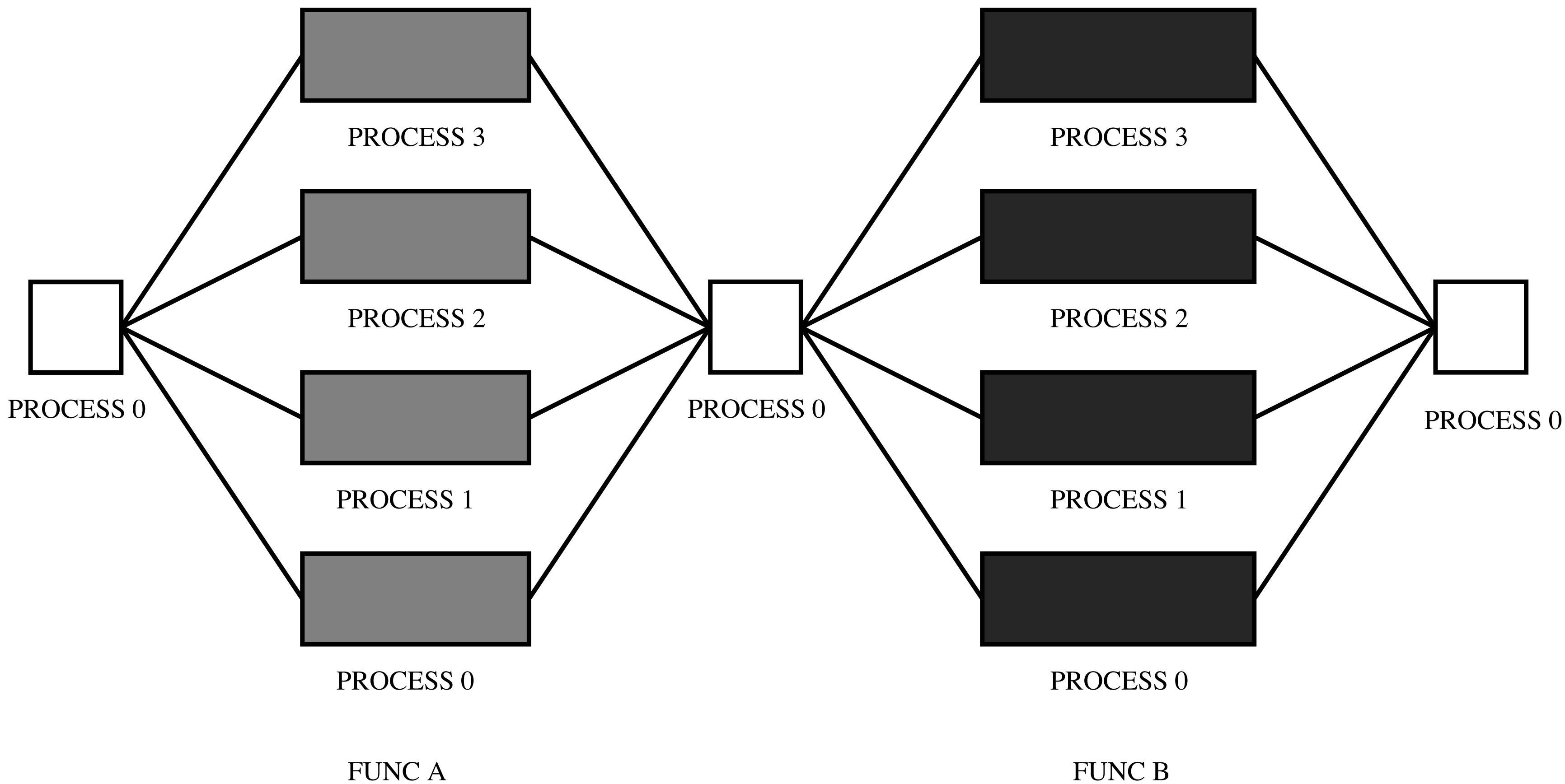


Process Based Parallel Program Model

- Processes executing in a data parallel fashion

```
main() {
  m_set_procs(4);
  m_fork(funcA);
  m_park_procs();
    ...
  m_rele_procs();
  funcB();
  m_kill_procs();
}
```

Process Based Parallel Program Model



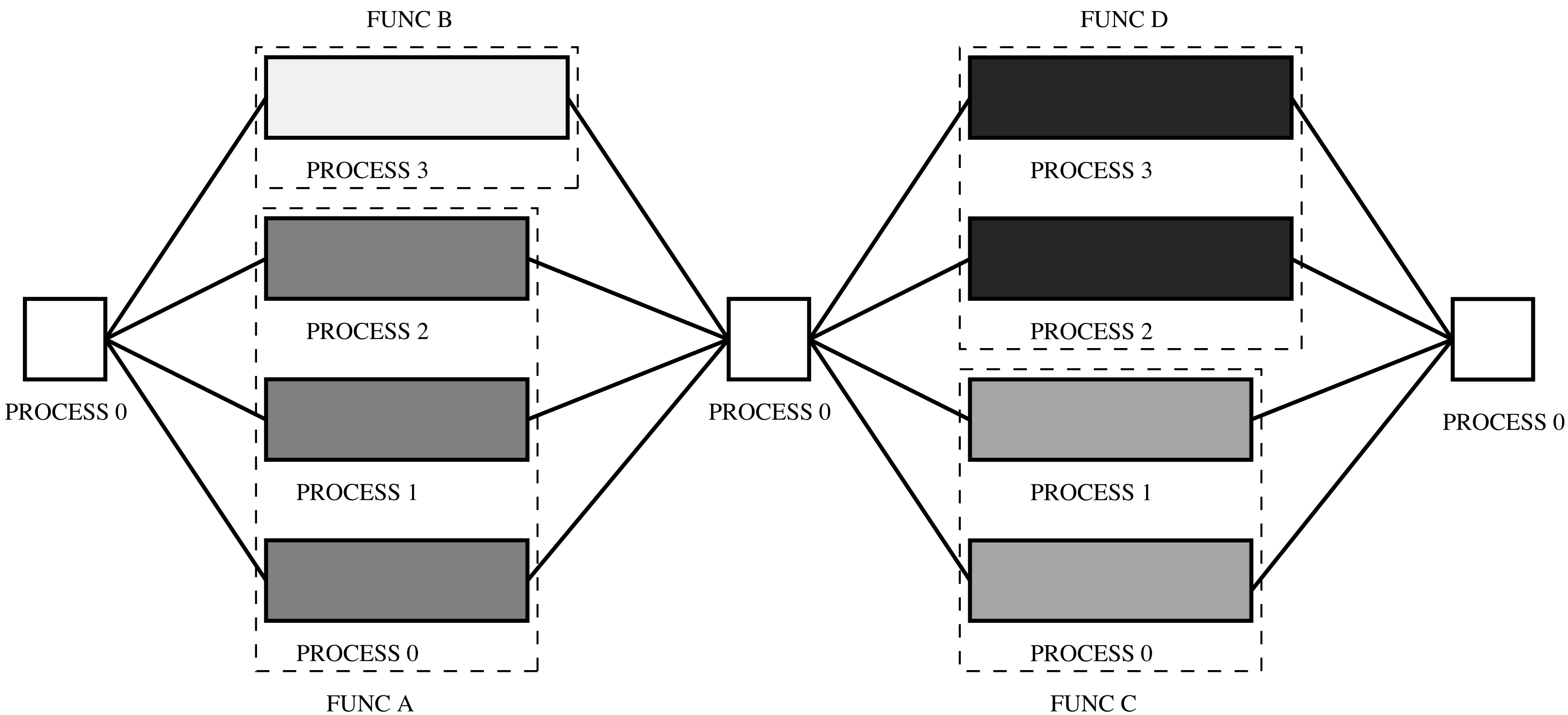
Process Based Parallel Program Model

- Processes executing in a function and data parallel fashion

```
main() {
  m_set_procs(4);
  m_fork(func1)
  m_park_procs();
     ...
  m_rele_procs();
  func2;
  m_kill_procs();
}
```

```
void func1() {
  switch (id) {
    case 0,1,2 : funcA();
 break;
    case 3 : funcB();
     break;
  }
}
void func2() {
  switch (id) {
    case 0,1 : funcC();
       break;
    case 2,3 : funcD();
       break;
  }
}
```

Process Based Parallel Program Model



Process Based Parallel Program Model

- Data parallel programming dominates
- Loops in programs are the source of data parallelism
- Explotation of parallelism involves sharing work in loops among processes
- Have to use appropriate scheduling techniques for optimal work sharing
- Parallelism in loops not always straightforward to find due to dependence
- Have to perform some transformations to expose parallelism

Summary

- An overview of shared memory parallel programming
- Process model: creation and destruction
- Shared variables
- Synchronization: locks, barriers
- Parallel programming basics
- NEXT CLASS: Shared Memory Parallel Programming - II
- READING: Kumar, Chap. 7